Towards Easing the Diagnosis of Bugs in OS Code

Henrik Stuart René Rydhof Hansen*
Julia L. Lawall Jesper Andersen
DIKU, University of Copenhagen,
Copenhagen, Denmark
{hstuart,rrhansen,julia,jespera}@diku.dk

Yoann Padioleau* Gilles Muller
OBASCO Group, École des Mines de
Nantes-INRIA, LINA, Nantes, France
{Yoann.Padioleau,Gilles.Muller}@emn.fr

Abstract

The rapid detection and treatment of bugs in operating systems code is essential to maintain the overall security and dependability of a computing system. A number of techniques have been proposed for detecting bugs, but little has been done to help developers analyze and treat them. In this paper we propose to combine bug-finding rules with transformations that automatically introduce bug-fixes or workarounds when a possible bug is detected. This work builds on our previous work on the Coccinelle tool, which targets device driver evolution.

1. Introduction

Recent years have seen a surge of interest in the development of bug-finding tools that are applicable to operating systems code [4, 6, 11, 12]. One of the first such tools was Engler et al.'s *Metal* [4], which used a control-flow based analysis, tempered by some pragmatic choices, to find bugs in Linux and OpenBSD. The choices made by Metal and similar tools to provide scalability, however, have in practice implied that such tools generate many false positives (reported code fragments that are not actually bugs) and may misidentify the source of a problem. The resulting bug reports thus have to be carefully checked by a maintainer. This is often a difficult task, both because of the large number of bug reports and because of the subtlety of operating systems code. These issues delay the deployment of protective and corrective actions, ultimately reducing the practical impact of bug-checking tools.

In previous work, we have considered the problem of changing device driver code automatically to reflect changes in the underlying Linux kernel driver APIs between different versions [17, 18, 19]. For this, we have developed the Coccinelle tool, which provides a language for describing code evolutions and a transformation engine for applying these evolutions automatically. The language is close to C code, but describes control-flow paths, as used by Metal. Based on this similarity, we have thus conjectured that Coccinelle could also be used for bug detection. Coccinelle, however, also allows easily specifying code transformations. This makes it possible to add bug fixes or logging code to matches found during the bug finding process. When a bug fix is possible, the system is protected immediately, with no user intervention. Logging code can help the maintainer identify real bugs more quickly, by providing access to information about the dynamic behavior of the system that is not available to static bug-finding tools. Furthermore, as recent research suggests [20], inserting application specific goal-oriented logging code can help improve forensic analysis of a breached system. Both techniques can thus help improve the security and dependability of operating systems.

In this paper we investigate how Coccinelle can be used as a flexible and powerful tool for bug finding (Section 2) and how the program transformation capabilities of Coccinelle can be leveraged to automatically insert or change code for proactive handling of a (potential) bug (Section 3). This handling can include workaround code that fixes the bug, prints useful logging/debugging messages, safely shuts down the system, prevents the bug, mitigates the consequences of the bug, or whatever is needed to handle a particular case. Such functionality is particularly useful in situations where a security vulnerability has been discovered but no official patch has yet been released. We then provide preliminary benchmarks comparing the preciseness of Coccinelle to the results of Engler et al. [4] (Section 4) and discuss work in progress to improve the preciseness and flexibility of our approach (Section 5). Finally, we present some related work (Section 6) and conclude (Section 7).

2. Finding Bugs with Coccinelle

Coccinelle provides a language, *Semantic Patch Language* (SmPL), for describing code transformations. A semantic patch has the form of a traditional Linux patch: it is based on C syntax and the code to be removed and added is designated with `-` and `+`, respectively. Nevertheless, a semantic patch is independent of whitespace and newlines, can abstract over subterms via a collection of *metavariables*, and can abstract over arbitrary control-flow paths via the "`...`" notation [18]. Figure 1 shows a simple example of a semantic patch implementing part of an evolution to Linux drivers identified in our previous work [19]. This semantic patch adds `GPF_ATOMIC` as the second argument to `usb_submit_urb` whenever this function is called with interrupts disabled.

In the rest of this section, we consider some of the bugs for which Metal patterns have been published, and show how these patterns can be expressed using SmPL.

* Hansen is now at Aalborg University, Padioleau is now at the University of Illinois, Urbana-Champaign.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLOS '07 October 18, 2007, Stevenson, Washington, USA.
Copyright © 2007 ACM 978-1-59593-922-7/07/0010...$5.00

```
@@ expression E; @@
  cli();
  ...    WHEN != ( sti(); | restore_flags(...); )
- usb_submit_urb(E)
+ usb_submit_urb(E, GFP_ATOMIC)
```

Figure 1. Evolving uses of `usb_submit_urb`

```
1 @@ @@
2   cli();
3   ... WHEN != ( sti(); | restore_flags(...); )
4 ? cli();
5
6 @@ @@
7 ( sti(); | restore_flags(...); )
8   ... WHEN != cli();
9 ( sti(); | restore_flags(...); )
```

Figure 2. Detecting bugs in interrupt status management

```
@@ @@
  cli();
  ... WHEN != ( sti(); | restore_flags(...); )
  kmalloc(...,GFP_KERNEL);
```

Figure 3. Calling `kmalloc` with interrupts disabled

Double enabling or disabling of interrupts An operating system relies on interrupts in order to be informed of certain asynchronous conditions. Leaving interrupts disabled for too long hurts the interactivity of the system, and disabling them forever can lead to a deadlock. The criticality of interrupt status management means that changes in the interrupt status are typically only done intraprocedurally; for example, if interrupts are enabled at the beginning of a function, they should be enabled when leaving the function as well. Functions that do not fit this pattern are thus suspicious. To find such potential bugs, we search for control-flow paths where:

1. Interrupts are disabled but not enabled before the end of the function or before again disabling them.
2. Interrupts are enabled twice, without being disabled in between.

Figure 2 shows a SmPL semantic patch that searches for control flow paths satisfying these conditions. To simplify the presentation, we have assumed like Engler et al. that interrupts are disabled using `cli` and enabled using either `sti` or `restore_flags`. The semantic patch can easily be extended to take into account a wider variety of interrupt status management functions, as we have done in the experiments described in Section 4. The semantic patch consists of two rules, each beginning with a pair of `@@`s. The first rule (lines 1-4) detects either a call to `cli` that is followed by a call to `cli` with no intervening call to `sti` or `restore_flags`. The question mark at the beginning of line 4 makes the detection of a second call to `cli` optional, meaning that the pattern also detects the case where there is a call to `cli` with no subsequent call to `cli`, `sti` or `restore_flags` before the end of the function. The second rule (lines 6-9) similarly detects successive calls to `sti` or `restore_flags` with no intervening call to `cli`. These rules thus detect all occurrences of the above conditions.

Blocking Calls. Another common source of bugs related to interrupt status management is calling a possibly blocking function in a context where interrupts are disabled. If

```
@@ expression E; @@
  kfree(E)
  ...
  E
```

Figure 4. SmPL patch for matching use-after-free bugs

```
@@ @@
  cli();
  ... WHEN != ( sti(); | restore_flags(...); )
+ warn("Double disable of interrupts in %s", __FUNCTION__);
  cli();

@@ @@
( sti(); | restore_flags(...); )
  ... WHEN != cli();
+ warn("Double enable of interrupts in %s", __FUNCTION__);
( sti(); | restore_flags(...); )
```

Figure 5. Warning about bugs in interrupt status management

```
@@ expression e; @@
  cli();
  ... WHEN != ( sti(); | restore_flags(...); )
- kmalloc(e,GFP_KERNEL);
+ kmalloc(e,GFP_ATOMIC);
```

Figure 6. Changing `kmalloc` flag

the function call actually blocks, the kernel is effectively deadlocked since no interrupts will be received to awaken the sleeping kernel. One such function is the kernel memory allocation function, `kmalloc`, which may block to wait for more memory to be available if it is given `GFP_KERNEL` as its second argument. Thus, `kmalloc` should not be called with this argument when interrupts are disabled. Figure 3 shows a semantic patch that searches for this pattern.

More Bugs. In addition to the interrupt status management bugs discussed above, we have implemented Coccinelle versions of all the Linux-relevant Metal scripts provided in [4]. Of particular interest are bugs related to memory management since they often have high impact on security and dependability. Figure 4 shows a semantic patch that searches for a (potential) use of a region of memory after it has been `free`'d. The repeated use of the metavariable `E` ensures that the argument to `kfree` is the same as some subsequent expression. Since Coccinelle uses syntactic matching, however, it will generate a false positive if there is an intervening reassignment of `E`. We consider how to reduce the number of false positives in Section 5. We have also implemented a semantic patch for detecting cases where the result of `kmalloc` is used without first being checked for `NULL`.

3. Fixing Bugs with Coccinelle

Our goal is not only to find potential bugs, but also to insert "workarounds", *i.e.*, code that is meant as a temporary stopgap, not as a permanent solution, to a potential problem. A workaround could disable a vulnerable service, provide extra logging to make forensic analysis easier, etc. By following the example of Figure 1, we can use Coccinelle to add such code during the bug-finding process.

We first consider the bugs related to the double enabling or disabling of interrupts. The semantic patch in Figure 2 detects that some control-flow path contains, *e.g.*, a double `cli` or double `sti`. Because these operations may be involved in other control-flow paths, we cannot simply remove the second one. Instead, we introduce logging code, to make any bugs that potentially involve a double `cli` or `sti` easier to diagnose. This transformation is performed automatically by the augmented semantic patch shown in Figure 5. The semantic patch could be further augmented to declare a flag in the enclosing function, to set and clear this flag on reaching a `cli` or `sti` respectively, and to only generate a warning or error when this flag indicates that a use of `cli` or `sti` is invalid. While such flags can be expensive to maintain, in this case, the semantic patch statically ensures that they are only added to functions in which there is a potential problem.

For calls to `kmalloc` that may potentially block with interrupts disabled, a more proactive approach is possible: changing `GFP_KERNEL` to `GFP_ATOMIC` will prohibit the kernel from blocking during memory allocation. Figure 6 shows a semantic patch that automatically performs this transformation. This transformation, however, is not a panacea since the fact that interrupts are disabled still prevents the kernel from allowing other processes to free up memory and thus will lead to more "out of memory" errors. Thus, the maintainer should ultimately consider what should be done at each code site. Still, the transformation is safe (with respect to deadlocking the kernel), and thus represents an appropriate workaround.

4. Preliminary Results

Engler et al. provide a web page with the results of applying their checkers to a variety of Linux kernels [15]. To validate our implementations of the above rules, we have repeated their experiments on Linux 2.3.99-pre6, which is the version for which results are provided on the web page that is closest to the version used in Engler et al.'s paper [4]. All of our experiments were carried out on a 3.40GHz Pentium 4. For each checker, the total processing time for all of the C files in the Linux 2.3.99 kernel source tree is under 15 minutes. In order to compare to Engler et al.'s work we only search for bugs and do not perform any program transformations. The code sites identified by these searches are of course candidates for rewriting.

Engler et al.'s results are classified as BUG or UNCHECKED. The former have been manually checked to be likely bug sites, while the latter may contain false positives. We thus focus on the matches classified as BUG. Furthermore, due to the large number of BUG null pointer errors, we consider only those derived from an allocation using the standard memory allocation function `kmalloc`. Table 1 indicates the number of checked bugs found by Metal, and the percentage of these that are found by Coccinelle.

Among the bugs that are not found by the current version of Coccinelle, the main problems are that the enclosing function could not be parsed, in which case Coccinelle skips to the next function, or a specialised macro was used ("naming" in Table 1), obscuring the source code. The parsing errors are primarily due to the fact that Coccinelle does not invoke the C preprocessor since this would make it impossible to detect bugs in code that disappears due to `#ifdef` processing. Specialised macros are used in some drivers to define their own memory allocation functions or macros, to override or extend `kmalloc`. Despite this, Coccinelle finds up to 100% of the Metal-detected bugs. Furthermore, our design decisions for Coccinelle imply that it finds some bugs that are overlooked by Metal. For example, in `drivers/net/3c515.c`, Coccinelle finds bugs in both branches of an `#ifdef`, but the

	detected	% detected	Missed bugs reasons	
	by Metal	by Coccinelle	Parsing	Naming
Interrupt checking	23	91%	9%	0%
Use of freed memory	17	100%	0%	0%
Deref of null pointers (kmalloc bugs only)	40	85%	8%	8%

Table 1. The number of bugs detected by Metal, and their treatment by Coccinelle

Interrupt checking: Double `sti()`	1
Interrupt checking: Error-paths with no `sti()`	1
Use of freed memory	1
Dereference of null pointers	3

Table 2. Bugs detected by Coccinelle, but overlooked by Metal

	Coccinelle	Metal [4] (Linux 2.3.99)
Use of freed memory	9 (53%)	43%
Dereference of null pointers	5(12%)	11%

Table 3. False positives. Percentages compare false positives to checked bugs.

```
void isdn_ppp_cleanup(void) {
  int i;
  for (i = 0; i < ISDN_MAX_CHANNELS; i++)
    kfree(ippp_table[i]);
}
```

Figure 7. Bug-free use of "freed" memory in drivers/isdn/isdn_ppp.c (Linux 2.3.99-pre6)

Metal results include only one of them. Table 2 indicates the number of such bugs found by Coccinelle, but not Metal.

Finally, like Metal, Coccinelle finds a number of false positives. Table 3 assesses the match sites that occur within the BUG files but are not marked as bugs by Engler et al. Our false positive rates within these files are roughly comparable to the false positive rates reported by Engler et al. [4].

5. Towards reducing the number of false positives

The rule detecting the use of freed memory finds all of the bugs identified by Engler et al., but has the highest rate of false positives among our examples. The source of these false positives is the case where some subterm of the argument to `kfree` is reassigned before the identified use of the argument expression. An example is shown in Figure 7. In this code, there is a control-flow path from the call to `kfree(ippp_table[i])` back to itself, and thus to a reference to its argument expression `ippp_table[i]`. This reference to the argument expression is reported as a bug by Coccinelle, but it is not a bug because the expression depends on the variable `i`, whose value changes on each iteration.

The problem here is that the current version of Coccinelle has no notion of data flow. That is, it can detect that two terms have the same structure, but it does not know whether they may have the same value. Staying within the current capabilities of Coccinelle, we could augment the semantic patch of Figure 4 by enumerating all the possible forms that the argument to `kfree` can have, and extending the `when` clause between the call to `kfree` and the use to check that none of the possible subexpressions are reassigned. The great variety of the possible forms of the argument to `kfree`, *e.g.*,

```
@@ expression E1, E2; @@
kfree(E1)
...
E2  where E1 = E2 and E1.dfa = E2.dfa
```

Figure 8. SmPL patch for matching use-after-free bugs

an array reference, a field reference, a field reference from an array reference, etc., however, implies that this approach is highly impractical. Instead, we need to be able to reason about arbitrary data-flow information, transparently, just as Coccinelle currently does for control-flow information.

To this end, we propose to extend Coccinelle with data-flow analysis capabilities, and provide these to SmPL via a new `where`-clause, so that we can filter away bug-free matches such as the one in Figure 7. Figure 8 shows an initial proposal for how to express this kind of filtering. In this example, we match a call to `kfree` with `E1` as an argument expression followed by an arbitrary expression `E2`, but constrain the matches so that `E1` and `E2` are syntactically equal and `E1` and `E2` are determined to be the same by data-flow analysis. These constraints are specified using the notation `E1 = E2` and the notation `E1.dfa = E2.dfa`, respectively.

Data-flow information encompasses not only use-def information, as used in the previous example, but also information about the range of concrete values that a particular expression can assume. Such information could be useful in detecting bugs such as buffer overflows, where each array reference needs to be compared to the size of the array. Detecting such bugs, however, typically involves arithmetic computations and constraint solving, which are outside the scope of Coccinelle. Instead we propose to extend SmPL with a general interface to integrate scripting language engines into Coccinelle, in order to be able to accept or reject matches based on more complex rules. Our goal is that using such scripting language engines should be transparent to Coccinelle, to make it easy to plug in any tool that could be relevant to a program searching or transformation task.

Figure 9 illustrates our proposal to interface Coccinelle with a scripting language, here Python. The semantic patch consists of two rules, one written in SmPL and one invoking Python. The SmPL code matches an array declaration and any subsequent references to it. The Python code is then invoked for each such match, based on the bindings indicated in the header (between the two occurrences of `@@`). The script uses a library, `cocci_lib`, that provides callbacks into Coccinelle, including access to the data-flow analysis engine. The `eval` function defined in this library first performs the data-flow analysis, if needed, memoizes the result, and then determines the set of possible values of the term in its argument. The `include_match` function defined in this library decides whether this instance of the match should be accepted as a possible bug or discarded as a false positive.

We consider the application of this semantic patch to the code shown in Figure 10, which includes the declaration and use of an array. On this code, the Python script in Figure 9 first calls `eval` to compute the size of the array, which will always return one value, as the size of the array must be a constant, and then uses `eval` again to compute all possible values that the array is dereferenced with, obtaining the range of values $[0; 20]$. Finally, the comparison in the last line evaluates to `True` as one element of the range is 20, which is outside the array's bounds. The value `True` is passed to `include_match`, indicating that the matched code represents a bug.

```
@@
identifier I;
expression E;
constant C;
type T;
@@
T I[C];
<... I[E] ...>

@@ script: python: C as x, E as y @@
buffer_size = cocci_lib.dfa.eval(x)[1]
index_values = cocci_lib.dfa.eval(y)
cocci_lib.include_match(max(index_values) >= buffer_size)
```

Figure 9. SmPL patch detecting potential buffer overflows

```
char buf[20];
int i;

for (i = 0; i <= 20; ++i)
  buf[i] = 'A';
```

Figure 10. Buffer overflow bug

6. Related Work

In recent years a number of bug-finding tools have been developed. Tools like Flawfinder [26] and RATS [21] use simple rules for syntactic searches whereas Metal [4] and Microsoft's SDV [1] use more elaborate semantics-based approaches. None of these tools provides facilities for program transformation.

Weimer has also proposed to address the difficulty of using automated bug-finding tools [25]. His approach combines each bug report with a possible bug fix, inferred from the difference between the existing code and a description of correct protocol usage. The bug fix must still be applied manually. Our approach automates the introduction of workaround code, but the lack of human intervention implies that the behavior of this code must be conservative, as illustrated by the examples in Section 3. It is thus still desirable for the maintainer to analyze the bug reports and make the most appropriate bug fix in each case, and Weimer's technique would be useful in this process.

A number of other frameworks for specifying and applying program transformations have recently been developed, from CIL [16] and XTC [7] that mainly support low-level rewriting of abstract syntax trees, to higher level languages such as JunGL [23] for refactoring C# programs and Stratego [24] for generic program transformation. While these languages are more expressive than SmPL, they are also more verbose, and thus do not allow the concise expression of the pattern to search for and the transformation to perform.

Aspect-Oriented Programming (AOP) has been used to extend OS code with new functionalities [2], including logging [13], and adding workaround code at run time [14]. While the pointcut languages of typical aspect systems only allow adding advice to individual operations [10], there has been some investigation of pointcut languages that allow describing sequences of operations [3], as we have used here. Typical aspect systems also only allow adding advice to high-level operations, such as method calls and field references, whereas SmPL allows matching against and adding or removing code around any C language element. We conjecture that the latter degree of expressiveness will be necessary for detecting some kinds of bugs, such as buffer overflows.

The Broadway compiler [8] has also adopted a pattern-based search language, but targeting a different area of application, namely program optimization. As part of their work they have developed and employed several data-flow analyses [9, 22] that we will investigate more closely as part of our work to reduce the number of false positives.

7. Conclusion

In this paper, we have proposed to improve the effectiveness of bug-finding tools by augmenting bug-finding rules with transformations that describe how to add workaround code to potential bug sites. This workaround code can provide a stop-gap solution between the time when the bug-finding tool is run and the time when the maintainer has completed the analysis and treatment of the generated bug reports. We have tested our bug finding and transformation tool on the set of examples defined in the initial paper on Metal [4], and we have found that our approach scales well to operating systems code and that we achieve comparable rates of bug detection and false positives. Finally, we have identified some areas in which our approach could be more expressive or flexible, both to reduce the number of false positives and to increase the range of bugs that can be identified.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In Eurosys'06 [5], pages 73–85.

[2] Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003*, pages 50–59, Boston, Massachusetts, Mar. 2003.

[3] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, Chicago, IL, USA, Mar. 2005.

[4] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[5] *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, Leuven, Belgium, Apr. 2006.

[6] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 38–51, Berlin, Germany, June 2002.

[7] R. Grimm. XTC: Making C safely extensible. In *Workshop on Domain-Specific Languages for Numerical Optimization*, Argonne National Laboratory, Aug. 2004.

[8] S. Z. Guyer and C. Lin. Optimizing the use of high performance software libraries. In *LCPC '00: Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, pages 227–243, London, UK, 2001. Springer-Verlag.

[9] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *SAS '03: Proceedings of the 10th International Static Analysis Symposium*, pages 214–236, San Diego, CA, USA, 2003. Springer-Verlag.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001.

[11] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. R. Engler. From uncertainty to belief: Inferring the specification within. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 161–176, Seattle, WA, USA, Nov. 2006.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, San Fransisco, CA, Dec. 2004.

[13] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In Eurosys'06 [5], pages 191–204.

[14] N. Loriant, M. Ségura-Devillechaise, and J.-M. Menaud. Server protection through dynamic patching. In *Proceedings of the 11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 343–349, Changsha, Hunan, China, Dec. 2005.

[15] Meta-level compilation. http://metacomp.stanford.edu/.

[16] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, 11th International Conference, CC 2002*, number 2304 in Lecture Notes in Computer Science, pages 213–228, Grenoble, France, Apr. 2002.

[17] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In *PLOS 2006: Linguistic Support for Modern Operating Systems*, pages 55–60, San Jose, CA, Oct. 2006.

[18] Y. Padioleau, J. L. Lawall, and G. Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. In *International ERCIM Workshop on Software Evolution (2006)*, Lille, France, Apr. 2006. Extended version available in Electronic Notes in Theoretical Computer Science, volume 166, pages 47-62, January 2007.

[19] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, Leuven, Belgium, Apr. 2006.

[20] S. Peisert, M. Bishop, S. Karin, and K. Marzullo. Towards models for forensic analysis. In *Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 3–15. IEEE, Apr. 2007.

[21] Secure Software, Inc. Web page: `http://www.securesw.com/download_rats.htm`.

[22] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th International Conference, CC 2006*, volume 3923 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2006.

[23] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *International Conference on Software Engineering (ICSE)*, pages 172–181, Shanghai, China, May 2006.

[24] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, 2004.

[25] W. Weimer. Patches as better bug reports. In *Proc. of Generative Programming and Component Engineering, GPCE'06*, pages 181–190, Portland, Oregon, USA, Oct. 2006.

[26] D. Wheeler. Flawfinder home page. Web page: `http://www.dwheeler.com/flawfinder/`, Oct. 2006.